

orthogonalization on a general purpose graphics processing unit with double double and quad double arithmetic

Jan Verschelde and Genady Yoffe

Department of Mathematics, Statistics, and Computer Science

University of Illinois at Chicago

851 South Morgan (M/C 249)

Chicago, IL 60607-7045, USA

Emails: jan@math.uic.edu and gyoffe2@uic.edu

URLs: www.math.uic.edu/~jan and www.math.uic.edu/~gyoffe

2 October 2012

Abstract

Our problem is to accurately solve linear systems of modest dimensions (typically, the number of variables equals 32) on a general purpose graphics processing unit. The linear systems originate from the application of Newton's method on polynomial systems of (moderately) large degrees. Newton's method is applied as a corrector in a path following method, so the linear systems are solved in sequence and not simultaneously. One solution path may require the solution of thousands of linear systems. In previous work we reported good speedups with our implementation to evaluate and differentiate polynomial systems on the NVIDIA Tesla C2050. Although the cost of evaluation and differentiation often dominates the cost of linear system solving, because of the limited bandwidth of the communication between CPU and GPU, we cannot afford to send the linear system to the CPU for solving.

Because of large degrees, the Jacobian matrix may contain extreme values, requiring extended precision, leading to a significant overhead. This overhead of multiprecision arithmetic is an additional motivation to develop a massively parallel algorithm. To allow overdetermined linear systems we solve linear systems in the least squares sense, computing the QR decomposition of the matrix by the modified Gram-Schmidt algorithm. We describe our implementation of the modified Gram-Schmidt orthogonalization method for the NVIDIA Tesla C2050, using double double and quad double arithmetic. Our experimental results show that the achieved speedups are sufficiently high to compensate for the overhead of one extra level of precision.

Key words and phrases. double double arithmetic, general purpose graphics processing unit, massively parallel algorithm, modified Gram-Schmidt method, orthogonalization, quad double arithmetic, quality up.

1 Introduction

Homotopy continuation methods provide reliable numerical algorithms to compute all isolated complex solutions of a polynomial system. A homotopy is a family of polynomial systems, connecting the system that has to be solved to a system with known solutions. Numerical continuation or predictor-corrector methods are applied to track the solution paths starting at the known solutions to the solutions of the system that has to be solve. Introductions to homotopy continuation methods are in [2] and [12]. For larger polynomial systems, it often happens that the double precision as provided by standard hardware is insufficient to achieve accurate results for some solution paths. In the problem setup for this paper we consider the tracking of one difficult solution path in extended precision.

The extended precision arithmetic we perform with the quad double library `QD 2.3.9` [7], and in particular on a GPU using the software at [14]. For the numerical properties, we refer to [5] and [19]. Our development of massively parallel algorithms is motivated by the desire to offset the extra cost of double double and quad double arithmetic. We strive for a quality up [1] factor: if we can afford to keep the execution time constant, by how much can we improve the quality of the solution?

Using double double or quad double arithmetic we obtain uniform running times. In [24] we experimentally determined that the overhead factors of double double over standard double arithmetic is indeed similar to the overhead of complex over standard double arithmetic. Relevant to quality, the errors are expected to decrease proportionally to the increase in the precision. In [23] we described a multicore implementation of a path tracker and the methods used to evaluate and differentiate systems of polynomials were implemented on the NVIDIA Tesla C2050 and described in [25]. The focus of this paper is on the solving of the linear systems, needed to run Newton's method.

Because of the limited bandwidth of CPU/GPU communication we cannot afford to transfer the evaluated system and its Jacobian matrix from the GPU to the CPU and perform the linear system solving on the CPU. Although the evaluation and differentiation of a polynomial system often dominates the cost of Newton's method [23], the cost of linear system solving increases relative to the parallel run times of evaluation and differentiation so that even with minor speedups, using a parallel version of the linear system solver matters in the overall execution time.

In the next section we state our problem, mention related work and list our contributions. In the third section we summarize the mathematical definition and properties of the modified Gram-Schmidt method and we illustrate the higher cost of complex multiprecision arithmetic. Then we describe our parallel version of the modified Gram-Schmidt algorithm and give computational results.

2 Problem Statement and Related Work

Our problem is to solve a linear system (which may have more equations than unknowns) on a GPU. The linear system occurs in the context of Newton's method applied to a polynomial system of modest size, e.g.: about 32 polynomials in 32 variables. The double precision as available in standard hardware is often insufficient to guarantee accurate results. Our goal is to offset the

extra cost of extended precision using a GPU.

Because the system could have more equations than unknowns and because of increased numerical stability, we decided to solve the linear system with a least squares method via a QR decomposition of the matrix. The algorithm we decided to implement is the modified Gram-Schmidt algorithm, see [8] for its definition and a discussion of its numerical stability.

Our main challenge for the parallel implementation of the modified Gram-Schmidt orthogonalization method is the modest size of the dimensions. In our experimental study we typically work with 32 variables, although the computational cost increases with a double digit factor if we use complex double double and complex quad double arithmetic.

2.1 Related Work

Comparing QR with Householder transformations and with the modified Gram-Schmidt algorithm, the authors of [18] show that on message passing systems, a parallel modified Gram-Schmidt algorithm can be much more efficient than a parallel Householder algorithm, and in any case not slower. MPI implementations of three versions of Gram-Schmidt orthonormalizations are described in [13]. The performance of different parallel modified Gram-Schmidt algorithms on clusters is described in [20]. Because the modified Gram-Schmidt method cannot be expressed by Level-2 BLAS operations, in [26] the authors proposed an efficient implementation of the classical Gram-Schmidt method.

In [16] is a description of a parallel QR with Gram-Schmidt on GPU and results on an implementation with the NVIDIA Geforce 295 are reported. A description on the performance of a high performance implementation of the QR algorithm on GPUs appeared in [9]. A report on QR decompositions on the NVIDIA Tesla C2050 can be found in [3]. The authors of [9] did not consider a fast implementation of the modified Gram-Schmidt method because the vectors in the inner products are large and the many synchronizations incur a prohibitive overhead. According to [9], a blocked version is susceptible to precision problems. In our setting, the length n of the vectors is small ($n = 32$ coincides with the warp size) and similar to what is reported in [3], we expect the cost of synchronizations to be modest for a small number of threads. Because of our small dimensions, we do not consider a blocked version.

In [4], the problem of solving many small independent QR factorizations on a GPU is investigated. Although our QR factorizations are also small, in our application of Newton's method in the tracking of one solution path, the linear systems are not independent and must be solved in sequence.

After the QR decomposition, we solve an upper triangular linear system. The solving of dense triangular systems on multicore and GPU accelerators is described in [21].

Related to polynomial system solving on a GPU, we mention two recent works. In [15], a subresultant method with a CUDA implementation of the FFT is described to solve systems of two variables. The implementation with CUDA of a multidimensional bisection algorithm on an NVIDIA GPGPU is presented in [11].

2.2 Our contributions

Based on our computational experiments, our conclusions are twofold:

1. Despite the low dimensions, we experimentally show that the extra cost of multiprecision arithmetic can be compensated by a GPU.
2. Combined with projected speedups of our massively parallel evaluation and differentiation implementation [25], the results pave the way for a path tracker that runs entirely on a GPU.

In this paper we apply the regular notion of speedup and compare the timings on the GPU with one core on the CPU for the *same* computations. When considering different levels of precision, a common practice is to compare the accuracy obtained at different precisions. Running independent calculations at different levels of precision is a pleasingly parallel computation which comes almost for free.

3 Modified Gram-Schmidt Orthogonalization

Roots of polynomial systems are typically complex and we calculate with complex numbers. Following notations in [6], the inner product of two complex vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ is denoted by $\mathbf{x}^H \mathbf{y}$, where \cdot^H denotes the Hermitian or conjugate transpose. In particular: $\mathbf{x}^H \mathbf{y} = \sum_{\ell=1}^n \bar{x}_\ell y_\ell$, where \bar{c} is the complex conjugate of $c \in \mathbb{C}$.

In order to describe our parallel implementation, in Figure 1 we state the pseudo code of the modified Gram-Schmidt orthogonalization method.

```

Input:  $A \in \mathbb{C}^{m \times n}$ .
Output:  $Q \in \mathbb{C}^{m \times n}$ ,  $R \in \mathbb{C}^{n \times n}$ :  $Q^H Q = I$ ,
         $R$  is upper triangular, and  $A = QR$ .
let  $\mathbf{a}_k$  be column  $k$  of  $A$ 
for  $k$  from 1 to  $n$  do
     $r_{kk} := \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$ 
     $\mathbf{q}_k := \mathbf{a}_k / r_{kk}$ ,  $\mathbf{q}_k$  is column  $k$  of  $Q$ 
    for  $j$  from  $k+1$  to  $n$  do
         $r_{kj} := \mathbf{q}_k^H \mathbf{a}_j$ 
         $\mathbf{a}_j := \mathbf{a}_j - r_{kj} \mathbf{q}_k$ 

```

Figure 1: The modified Gram-Schmidt orthogonalization algorithm.

Given the QR decomposition of a matrix A , the system $A\mathbf{x} = \mathbf{b}$ is equivalent to $QR\mathbf{x} = \mathbf{b}$. By the orthogonality of Q , solving $A\mathbf{x} = \mathbf{b}$ is reduced to the upper triangular system $R\mathbf{x} = Q^H \mathbf{b}$. This solution minimizes $\|\mathbf{b} - A\mathbf{x}\|_2^2$.

Instead of computing $Q^H \mathbf{b}$ in a separate stage, for numerical stability as recommended in [8, §19.3], one must apply the modified Gram-Schmidt method to the matrix A augmented with \mathbf{b} :

$$\begin{bmatrix} A & \mathbf{b} \end{bmatrix} = \begin{bmatrix} Q & \mathbf{q}_{n+1} \end{bmatrix} \begin{bmatrix} R & \mathbf{y} \\ 0 & z \end{bmatrix}. \quad (1)$$

As \mathbf{q}_{n+1} is orthogonal to the column space of Q , we have $\|\mathbf{b} - A\mathbf{x}\|_2^2 = \|R\mathbf{x} - \mathbf{y}\|_2^2 + z^2$ and $\mathbf{y} = Q^H \mathbf{b}$.

As reported in [8], the number of flops in the algorithm in Figure 1 equals $2mn^2$. The experimental results in Table 1 show the cubic behavior of the running time: doubling the dimension multiplies the user CPU time by a factor of (almost) eight.

Table 1: User CPU times for 10,000 QR decompositions with double arithmetic on n random vectors of dimension n , for increasing values of n .

n	CPU time	factor
32	3.7 sec	1.0
64	28.7 sec	7.8
128	225.4 sec	7.9
256	1787.1 sec	7.9

The experiments reported in this and the next section were done on one core of an 3.47 Ghz Intel Xeon X5690 and with the code in version 2.3.70 of PHCpack [22].

4 complex and multiprecision arithmetic: cost and accuracy

With user CPU times of runs with the modified Gram-Schmidt algorithm on random data in Table 2 we illustrate the overhead factor of using complex double, complex double double, and complex quad double arithmetic over standard double arithmetic. Considering absolute times, the 3.7 seconds increase to 788.3 seconds (more than 48 minutes) when going from double to complex quad double arithmetic.

Table 2: User CPU times for 10,000 QR decompositions on 32 random complex vectors of dimension 32, for increasing levels of precision.

precision	CPU time	factor
double	3.7 sec	1.0
complex double	26.8 sec	7.2
complex double double	291.5 sec	78.8
complex quad double	2916.8 sec	788.3

Using the cost factors we can recalibrate the dimension. Suppose a flop costs 8 times more, using $8 = 2^3$, the number of flops in the modified Gram-Schmidt method is then $8 \times 2mn^2 = 2(2m)(2n)^2$. Working with operations that cost 8 times more increases the cost with the same factor as doubling the dimension in the original arithmetic.

Taking the cubed roots of the factors in Table 2: $7.2^{1/3} \approx 1.931$, $78.8^{1/3} \approx 4.287$, $788.3^{1/3} \approx 9.238$, we can say that the cost of using complex double, complex double double, and complex quad double arithmetic is equivalent to working in double arithmetic, but multiplying the dimension of the problem respectively by the factors 1.931, 4.287, and 9.238. Multiplying the original dimension 32 with 1.931, 4.187, and 9.238 respectively yields 62, 134, 296. So we may say that orthogonalizing 32 vectors in \mathbb{C}^{32} in quad double arithmetic costs the same amount of work as orthogonalizing 296 vectors in \mathbb{C}^{296} .

Next we compare the running times of the modified Gram-Schmidt algorithm with the running times for the LU factorization. In [8, page 174], LU factorization with the method of Doolittle on an m -by- n matrix (for $m \geq n$) has cost $n^2(m - n/3)$ which for $m = n$ equals $2n^3/3$. Not taking the cost of arithmetic into consideration, LU factorization is three times faster than the modified Gram-Schmidt algorithm. In Table 3, we take the user CPU times of Table 2 and compare these times with the same number of LU factorizations, for the same dimension and also for different kinds of arithmetic.

Table 3: User CPU times for 10,000 QR decompositions with the modified Gram-Schmidt algorithm (MGS) and 10,000 LU factorizations on 32 random complex vectors of dimension 32, for increasing levels of precision.

precision	MGS time	LU time	factor
double	3.7 sec	2.6 sec	1.4
complex double	26.8 sec	9.1 sec	2.9
complex double double	291.5 sec	95.6 sec	3.1
complex quad double	2916.8 sec	920.1 sec	3.2

The last column of Table 3 is the time of the modified Gram-Schmidt algorithm over the time of the LU factorization. Except for the calculations in standard double precision, the factor averages 3 and increases as the precision increases. An explanation for this increase could be that the square root computation costs more for longer numbers.

Because the modified Gram-Schmidt algorithm takes three times longer than the LU factorization, the cost of linear system solving relative to the evaluation and differentiation of the polynomials grows significantly. This growth is significant enough that we can actually no longer regard the cost of linear system solving as being dominated by the evaluation and differentiation cost.

To measure the accuracy of the computed $Q \in \mathbb{C}^{m \times n}$ and $R \in \mathbb{C}^{n \times n}$ of a given $A \in \mathbb{C}^{m \times n}$, we consider the matrix 1-norm [6] of $A - QR$:

$$e = \|A - QR\|_1 = \max_{\substack{i=1,2,\dots,m \\ j=1,2,\dots,n}} \left| a_{ij} - \sum_{\ell=1}^n q_{i\ell} r_{\ell j} \right|. \quad (2)$$

For $x \in [-10, +10]$, we have $x^d \in [10^{-d}, 10^{+d}]$, so as the degrees d of the polynomials in our system increase we are likely to obtain more extreme values in the Jacobian matrix. In the experiments discussed below we generate complex numbers of modulus one as $\exp(i\theta)$, where $i = \sqrt{-1}$ and θ is chosen at random from $[0, 2\pi[$. To generate complex numbers of varying

magnitude, we consider $r \exp i\theta$, with r chosen at random from $[10^{-g}, 10^{+g}]$ where g determines the range of the moduli of the generated complex numbers. To simulate the numbers in the Jacobian matrices arising from evaluating polynomials of degree d , it seems natural to take the parameter g equal to d .

In Table 4 experimental values for e are summarized. For complex numbers with moduli in $[10^{-g}, 10^{+g}]$ we see that the decimal logarithm of e decreases linearly as g increases. The difference in the logarithm of the smallest and the largest error remains almost constant and increases slightly as g increases. For $g = 16$ the results are totally wrong with complex doubles, while with complex double double arithmetic we still obtain accurate answers of the usual double precision. For complex double double and complex quad double arithmetic, the experiment is continued in Table 5 for g ranging from 17 to 32. In this range, the precision of double double arithmetic becomes insufficient and quad double arithmetic is needed for accurate results.

Table 4: For 1,000 QR decompositions on 32-by-32 matrices with randomly generated complex numbers of magnitudes uniformly distributed in $[10^{-g}, 10^{+g}]$ and for g ranging from 1 to 16, we list $m_e = \min(\log_{10}(e))$, $M_e = \max(\log_{10}(e))$, and $D_e = m_e - M_e$, computed in complex double and complex double double arithmetic.

g	complex double			complex double double		
	m_e	M_e	D_e	m_e	M_e	D_e
1	-14.5	-14.0	0.5	-30.6	-30.1	0.5
2	-13.7	-13.0	0.7	-29.7	-29.1	0.6
3	-12.6	-12.1	0.5	-28.7	-28.1	0.7
4	-11.7	-11.0	0.7	-27.8	-27.1	0.7
5	-10.7	-10.1	0.7	-26.9	-26.1	0.8
6	-9.8	-9.1	0.7	-25.9	-25.0	0.8
7	-8.9	-8.0	0.9	-24.9	-24.1	0.8
8	-7.8	-7.0	0.8	-24.0	-23.1	1.0
9	-6.9	-6.1	0.8	-23.0	-22.2	0.8
10	-5.9	-5.0	1.0	-22.1	-21.1	1.0
11	-4.8	-4.1	0.7	-21.1	-20.1	1.0
12	-3.9	-3.1	0.8	-20.1	-19.2	0.9
13	-3.0	-2.0	1.0	-19.3	-18.2	1.1
14	-2.0	-1.0	1.0	-18.4	-17.2	1.2
15	-1.2	-0.1	1.1	-17.3	-16.2	1.1
16	-0.2	1.0	1.2	-16.4	-15.1	1.3

Our experiments show that the numerical stability of the modified Gram-Schmidt method is good and predictable. If we do not want to lose more than half of the number of decimal places when working with complex numbers ranging in modulus between 10^{-g} and 10^{+g} , then we must compute with a working precision of at least $2g$ decimal places.

Table 5: For 1,000 QR decompositions on 32-by-32 matrices with randomly generated complex numbers of magnitudes uniformly distributed in $[10^{-g}, 10^{+g}]$ and for g ranging from 17 to 32, we list $m_e = \min(\log_{10}(e))$, $M_e = \max(\log_{10}(e))$, and $D_e = m_e - M_e$, computed in complex double double and complex quad double arithmetic.

g	complex double double			complex quad double		
	m_e	M_e	D_e	m_e	M_e	D_e
17	-15.5	-14.1	1.3	-48.1	-47.1	1.0
18	-14.6	-13.2	1.4	-47.1	-46.2	1.0
19	-13.6	-12.0	1.5	-46.4	-45.1	1.2
20	-12.6	-11.1	1.5	-45.1	-44.2	0.9
21	-11.4	-10.2	1.2	-44.2	-43.2	1.0
22	-10.7	-9.1	1.5	-43.3	-42.2	1.1
23	-9.8	-8.1	1.7	-42.4	-41.1	1.3
24	-8.8	-7.2	1.6	-41.3	-40.2	1.2
25	-7.8	-6.3	1.5	-40.2	-39.2	1.0
26	-6.8	-5.2	1.6	-39.3	-38.1	1.1
27	-5.7	-4.2	1.5	-38.4	-37.2	1.2
28	-4.7	-3.2	1.5	-37.7	-36.1	1.6
29	-3.7	-2.2	1.5	-36.8	-35.2	1.6
30	-3.0	-1.2	1.8	-35.7	-34.2	1.5
31	-1.8	-0.2	1.6	-34.8	-33.3	1.5
32	-1.0	0.8	1.9	-33.9	-32.2	1.8

5 Massively Parallel Modified Gram-Schmidt Orthogonalization

Our main kernel `Normalize.Remove()` in Gram-Schmidt orthogonalization normalizes a vector and removes components of all vectors with bigger indexes in the direction of this vector. The secondary kernel `Normalize()` only normalizes one vector. The algorithm in Figure 2 overwrites the input matrix A so that on return the matrix A equals the matrix Q of the algorithm in Figure 1.

The multiple blocks launched by the kernel within each iteration of the loop in the algorithm in Figure 2 is the first coarse grained level of parallelism. If we keep in mind that the number of variables equals 32, the number of cores on a multiprocessor of the GPU, then the second fine grained parallelism resides in the calculation of componentwise operations and of the inner products. Threads within blocks perform these operations cooperatively. As one inner product of two 32-vectors requires 32 multiplications (one operation per core), note that a multiplication in double double and quad double arithmetic requires many operations with hardware doubles.

As the algorithm suggests the normalization of each \mathbf{a}_k is performed $(n - k)$ times — by each of the blocks in the k th launch of the kernel `Normalize.Remove()`. However normalizing it only once instead would suggest another launch of the kernel `Normalize()` associated with extra writing to and reading from the global memory of the card of the vector being normalized. This would be more expensive than to perform the normalization within `Normalize.Remove()` multiple

Input: $A \in \mathbb{C}^{m \times n}$, $A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \dots \ \mathbf{a}_n]$,
 $\mathbf{a}_k \in \mathbb{C}^m$, $k = 1, 2, \dots, n$.
 Output: $A \in \mathbb{C}^{m \times n}$, $A^H A = I$ (i.e.: $A = Q$),
 $R \in \mathbb{C}^{n \times n}$: $R = [r_{ij}]$, $r_{ij} \in \mathbb{C}$,
 $i = 1, 2, \dots, n$, $j = 1, 2, \dots, n$.
 for k from 1 to $n - 1$ do
 launch kernel **Normalize_Remove**(k)
 with $(n - k)$ blocks of threads,
 as the j th block (for all $j : k < j \leq n$)
 normalizes \mathbf{a}_k and removes the component
 of \mathbf{a}_j in the direction of \mathbf{a}_k
 launch kernel **Normalize**(n) with one
 thread block to normalize \mathbf{a}_n .

Figure 2: A parallel version of the modified Gram-Schmidt orthogonalization algorithm.

times.

The loop in the algorithm in Figure 2 performs $n - 1$ normalizations, where each normalization is followed by the update of all remaining vectors. In particular, after normalizing the k th vector, we launch $n - k$ blocks of m threads. Each thread block handles one \mathbf{a}_j . The update stage has a triangular structure. The triangular structure implies that we have more parallelism for small values of k . Therefore, we expect increased speedups at earlier stages of the algorithm in Figure 2.

The main ingredients in the kernels **Normalize**() and **Normalize_Remove**() are inner products and the normalizations, which we explain in the next two subsections. In subsection C we discuss the usage of the card resources by threads of the kernel **Normalize_Remove**().

5.1 computing inner products

The warp size of the Tesla C2050 equals 32, and 32 is our typical dimension. In computing $\mathbf{x}^H \mathbf{y}$ the products $\bar{x}_\ell \star y_\ell$ are independent of each other. The inner product $\mathbf{x}^H \mathbf{y}$ is computed in two stages:

1. All threads work independently in parallel: thread ℓ calculates $\bar{x}_\ell \star y_\ell$ where the operation \star is a complex double, a complex double double, or a complex quad double multiplication. Afterwards, all threads in the block are synchronized for the next stage.
2. The application of a sum reduction [10, Figure 6.2] to sum the elements in $(\bar{x}_1 y_1, \bar{x}_2 y_2, \dots, \bar{x}_m y_m)$ and compute $\bar{x}_1 y_1 + \bar{x}_2 y_2 + \dots + \bar{x}_m y_m$. The $+$ in the sum above corresponds to the \star in the item above and is a complex double, a complex double double, or a complex quad double addition. There are $\log_2(m)$ steps but as the m typically equals the warp size, there is thread divergence in every step.

The number of shared memory locations used by an inner product equals $2m$, times the space for a complex double, a complex double double, or a complex quad double.

The $2m$ memory locations suffice if we have to compute only one inner product and if we allow that one of the original vectors is overwritten. In our algorithm, we need the same vector \mathbf{q}_k multiple times when computing $r_{kj} := \mathbf{q}_k^H \mathbf{a}_j$ (see Figure 1) so we need an extra m shared memory locations to store $\bar{q}_{k\ell} \star a_{j\ell}$ for $\ell = 1, 2, \dots, m$. Then the computation of $\mathbf{a}_j := \mathbf{a}_j - r_{kj} \mathbf{q}_k$ requires another m memory locations to store the products $r_{kj} \star q_{k\ell}$ for $\ell = 1, 2, \dots, m$. So in total we have $4m$ memory locations in shared memory in the kernel `Normalize_Remove()`.

5.2 the orthonormalization stage

After the computation of the inner product $\mathbf{a}_k^H \mathbf{a}_k$, the orthonormalization stage consists in one square root computation, followed by m division operations.

The first thread in a block performs the square root calculation $r_{kk} := \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$ and then, after a synchronization, the m threads in a block independently perform the divisions $a_{k\ell} := a_{k\ell} / r_{kk}$, for $\ell = 1, 2, \dots, m$ to compute \mathbf{q}_k , stored in the location of \mathbf{a}_k .

Dividing each component of a vector by the norm happens independently, and as the cost of the division increases for complex doubles, complex double doubles, and complex quad doubles, so we could expect an increased parallelism as we increase the working precision. Unfortunately, also the cost for the square root calculation — executed in isolation by the first thread in each block — also increases as the precision increases.

5.3 the occupancy of multiprocessors

In the occupancy calculations, we take $m = n$, and consider the use of complex double double arithmetic. Concerning the occupancy of the multiprocessors, one thread block uses

$$4 \times n \times \text{size_of}(\text{complex double double}) \quad (3)$$

bytes of shared memory, which amounts to a bit more than 4000 bytes. Also a thread block uses about 3000 registers. The number of blocks scheduled per multiprocessor is 8. It is actually the maximum number of blocks which could be scheduled per multiprocessor for the device of compute capability 2.0. Neither allocated per block shared memory, neither the number of registers used do not appear as the limiting factor on the number of blocks scheduled per multiprocessor. Although shared memory and registers of a multiprocessor are employed quite well: 8 blocks of threads use about

$$100 \times 4,224 \times 8 / 49,152 \approx 68\% \quad (4)$$

of shared memory capacity, and

$$100 \times 3,072 \times 8 / 32,768 \approx 75\% \quad (5)$$

of available registers. For dimension 32, the orthogonalization launches the kernel `Normalize_Remove()` 31 times, while first 7 of these launches employ 4 multiprocessors, launches from 8 to 15 employ 3 multiprocessors, 16-23 employ 2 multiprocessors, and finally launches 24-31 employ only one multiprocessor.

5.4 data movement

At the beginning of the kernel thread ℓ of a block reads the ℓ th component of the vector \mathbf{a}_k from the global memory into the ℓ th location of the first column of the shared memory $4 \times m$ two dimensional array `Sh_Locations` of complex numbers of the given precision allocated by the block. Subsequently it reads the ℓ th component of the vector \mathbf{a}_j into the ℓ th location of the second column of `Sh_Locations`. Both readings are prescribed to be done simultaneously by threads of the block.

For double and double double precision levels we achieve coalesced access to the global memory but not for complex quad double numbers. This could explain why the speedups do not increase as we go from complex double double to the complex quad double versions of the parallel Gram-Schmidt algorithm.

6 the back substitution kernel

After the computation of Q and R , denoting $Q^H \mathbf{b}$ by \mathbf{y} , we have to solve the triangular system $R\mathbf{x} = \mathbf{y}$. Because of the low dimension of our application, only one block of threads will be launched. Pseudo code for a parallel version of the back substitution algorithm is shown in Figure 3.

The most natural order for the parallel version of the back substitution appears to be to process the matrix R in a column fashion. In the k th step we multiply the k th column of R by x_k and subtract the product from the right hand side vector \mathbf{y} updated by such subtractions at all the previous steps.

```

Input:  $R \in \mathbb{C}^{n \times n}$ , an upper triangular matrix,
        $\mathbf{y} \in \mathbb{C}^n$ , the right hand side vector.
Output:  $\mathbf{x}$  is the solution of  $R\mathbf{x} = \mathbf{y}$ .
for  $k$  from  $n$  down to 1 do
    thread  $k$  does  $x_k := y_k / r_{kk}$ 
    all threads synchronize
    for  $j$  from 1 to  $k - 1$  do
        thread  $j$  does  $y_j := y_j - r_{jk} \star x_k$ 
    all threads synchronize

```

Figure 3: Pseudo code for a parallel back substitution.

Ignoring the cost of synchronization and thread divergence and with the warp size 32 equal to the dimension n , the parallel execution reduces the inner loop to one step. With focus on the arithmetical cost, the total number of steps equals $2n$. Note that the more costly division operator is done by only one thread. More precisely than $2n$, the arithmetical cost of the algorithm in Figure 3 is n divisions, followed by n multiplications and n subtractions.

During the execution of the parallel back substitution, the right hand side vector \mathbf{y} remains in shared memory. At each step k , the current column k of R is loaded into shared memory for processing.

Because only one block can be occupied for this stage, running the same code at different levels of precision provides a free error estimate for the solution of the linear system.

7 Computational Results

Our computations were done on an HP Z800 workstation running Red Hat Enterprise Linux. For speedups, we compare the sequential run times on one core of an 3.47 Ghz Intel Xeon X5690. The clock speed of the NVIDIA Tesla C2050 is at 1147 Mhz, about three times slower than the clock speed of the CPU. The C++ code is compiled with version 4.4.6 of gcc and we use release 4.0 of the NVIDIA CUDA compiler driver [17].

We ran the modified Gram-Schmidt method on 32 random complex vectors of dimension 32. The times and speedups are shown in Table 6.

Table 6: Wall clock times and speedups for 10,000 orthogonalizations on 32 random complex vectors of dimension 32.

precision	1 CPU core	Tesla C2050	speedup
complex double	13.376 sec	5.339 sec	2.5
complex double double	115.63 sec	16.5 sec	7.0
complex quad double	785 sec	108 sec	7.3

In Table 6 compare the 115.63 seconds on 1 CPU core to perform the calculations in double double arithmetic with the 108 seconds to do the same computations on a GPU in quad double arithmetic. Using a GPU we achieve a quality up: in about the same time we obtain more accurate orthogonalizations.

Table 7: Wall clock times and overall speedups for 10,000 orthogonalizations on 32 random complex vectors of dimension 32 and for 10,000 polynomial evaluations and differentiations of polynomial system of 32 equations with 32 variables, with 32 monomials per polynomial, with 5 variables in each monomial, with variable degrees uniformly taken from $\{1, 2, 3, 4, 5\}$.

precision	CPU PE	GPU PE	speedup
complex double	11 sec	1.3 sec	8.5
complex double double	66 sec	2.1 sec	31.4
complex quad double	396 sec	14.2 sec	27.9
precision	CPU MGS	GPU MGS	speedup
complex double	13.4 sec	5.3 sec	2.5
complex double double	115.6 sec	16.5 sec	7.0
complex quad double	785.0 sec	108.0 sec	7.0
precision	CPU PE+MGS	GPU PE+MGS	speedup
complex double	24.4 sec	6.6 sec	3.7
complex double double	181.6 sec	18.6 sec	9.8
complex quad double	1181.0 sec	122.2 sec	9.7

Table 8: Wall clock times and overall speedups for 10,000 orthogonalizations on 32 random complex vectors of dimension 32 and for 10,000 polynomial evaluations and differentiations of polynomial system of 32 equations with 32 variables, with 32 monomials per polynomial, with 12 variables in each monomial, with variable degrees uniformly taken from $\{1, 2, \dots, 10\}$.

precision	CPU PE	GPU PE	speedup
complex double	22.5 sec	1.8 sec	12.5
complex double double	135.0 sec	4.1 sec	32.9
precision	CPU MGS	GPU MGS	speedup
complex double	13.4 sec	5.3 sec	2.5
complex double double	115.6 sec	16.5 sec	7.0
precision	CPU PE+MGS	GPU PE+MGS	speedup
complex double	35.9 sec	7.1 sec	5.1
complex double double	250.6 sec	20.6 sec	12.2

In Tables 7 and 8 projected overall speedups are computed for Tesla 2050 path tracking. We see that speedups are higher as there are more variables in the monomials and as the degrees of the variables are higher. Our polynomial evaluation and differentiation kernels have to be tuned so that their thread blocks would use more multiprocessor registers than shared memory for to be able to work with quad double arithmetic for systems of characteristics as in Table 8.

8 Conclusion

Using a massively parallel algorithm for the modified Gram-Schmidt orthogonalization on a NVIDIA Tesla C2050 Computing Processor we can compensate for the cost of one extra level of precision, even already for modest dimensions.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1115777.

References

- [1] S.G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.
- [2] E.L. Allgower and K. Georg. *Introduction to Numerical Continuation Methods*, volume 45 of *Classics in Applied Mathematics*. SIAM, 2003.
- [3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS 2011)*, pages 48–58. IEEE Computer Society, 2011.

- [4] M.J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebraic problems in GPU registers. In *Proceedings of the 2012 IEEE International Parallel Distributed Processing Symposium (IPDPS 2012)*, pages 2–13. IEEE Computer Society, 2012.
- [5] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
- [6] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.
- [7] Y. Hida, X.S. Li, and D.H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*, pages 155–162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996, software at <http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.9.tar.gz>.
- [8] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.
- [9] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In D. Kaeli and M. Leeser, editors, *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU’09)*, pages 71–78. ACM, 2009.
- [10] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann, 2010.
- [11] R.A. Klopotek and J. Porter-Sobieraj. Solving systems of polynomial equations on a GPU. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Preprints of the Federated Conference on Computer Science and Information Systems, September 9-12, 2012, Wroclaw, Poland*, pages 567–572, 2012.
- [12] T.Y. Li. Numerical solution of polynomial systems by homotopy continuation methods. In F. Cucker, editor, *Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics*, pages 209–304. North-Holland, 2003.
- [13] F.J. Linger. Efficient Gram-Schmidt orthonormalisation on parallel computers. *Communications in Numerical Methods in Engineering*, 16(1):57–66, 2000.
- [14] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010. Software at <http://code.google.com/p/gpugrpec/>.
- [15] M.M. Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *Journal of Physics: Conference Series*, 341, 2011. Proceedings of High Performance Computing Symposium 2011, Montreal, 15-17 June 2011.
- [16] B. Milde and M. Schneider. Parallel implementation of classical Gram-Schmidt orthogonalization on CUDA graphics cards. Available via <https://www.cdc.informatik.tu-darmstadt.de/de/cdc/personen/michael-schneider>.

- [17] NVIDIA. NVIDIA CUDA Programming Guide. Version 3.0. 2010.
- [18] D.P. O’Leary and P. Whitman. Parallel QR factorization by Householder and modified Gram-Schmidt algorithms. *Parallel Computing*, 16(1):99–112, 1990.
- [19] D.N. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992. <ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z>.
- [20] G. Rünger and M. Schwind. Comparison of different parallel modified Gram-Schmidt algorithms. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing (Euro-Par’05)*, volume 3648 of *Lecture Notes in Computer Science*, pages 826–836. Springer-Verlag, 2005.
- [21] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDSW 2010)*, pages 1–8. IEEE Computer Society, 2010.
- [22] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999. Software available at <http://www.math.uic.edu/~jan/download.html>.
- [23] J. Verschelde and G. Yoffe. Quality up in polynomial homotopy continuation by multi-threaded path tracking. Preprint [arXiv:1109.0545v1](https://arxiv.org/abs/1109.0545v1) [cs.DC] 2 Sep 2011.
- [24] J. Verschelde and G. Yoffe. Polynomial homotopies on multicore workstations. In M.M. Maza and J.-L. Roch, editors, *Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France*, pages 131–140. ACM, 2010.
- [25] J. Verschelde and G. Yoffe. Evaluating polynomials in several variables and their derivatives on a GPU computing processor. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops. 21-25 May 2012, Shanghai China*, pages 1391–1399. IEEE Computer Society, 2012.
- [26] T. Yokozawa, D. Takahashi, T. Boku, and M. Sato. Efficient parallel implementation of classical Gram-Schmidt orthogonalization using matrix multiplication. In *Parallel Matrix Algorithms and Applications (PMAA 2006)*, pages 37–38, 2006.